LANGLEY
GRANT

IN-61-CR

200668
36P

PARALLEL PROCESSORS AND NONLINEAR STRUCTURAL DYNAMICS ALGORITHMS AND SOFTWARE

Principal Investigator: Ted Belytschko

Department of Civil Engineering
Northwestern University
Evanston, Illinois 60208-3109

Semiannual Progress Report

March 1, 1988 though August 31, 1988

NASA Research Grant NAG-1-650

Vectorized, Concurrent Finite Element Program

1 Introduction

A nonlinear structural dynamics finite element program has been developed to run on a shared-memory multiprocessor with pipeline processors. The program WHAMS [1] was used as a framework for this work. The program employs explicit time integration and has the capability to handle both the nonlinear material behavior and large displacement response of three dimensional structures. The elasto-plastic material model, described in [2], uses an isotropic strain hardening law which is input as a piecewise linear function. Geometric nonlinearities are handled by a corotational formulation in which a coordinate system is embedded at the integration point of each element. Currently, the program has an element library consisting of a beam element based on Euler-Bernoulli theory and triangular and quadrilateral plate elements based on Mindlin theory.

2 Explicit Finite Element Formulation

2.1 Finite Element Equations

The equations of motion for a structural system are given by:

$$\boldsymbol{M}\,\boldsymbol{a} + \boldsymbol{f}_{int} = \boldsymbol{f}_{ext} \tag{1}$$

where,

$\boldsymbol{M}$	=	global mass matrix,
$\boldsymbol{a}$	=	nodal accelerations,
$\boldsymbol{f}_{int}$	=	assembled internal nodal forces,
$\boldsymbol{f}_{ext}$	=	assembled external nodal forces.

The mass matrix is assumed to be diagonal and lumped so that the system equations are uncoupled. The internal nodal force is computed on the element level by

$$\boldsymbol{f}^{e}_{int} = \int_{\Omega^e} \boldsymbol{B}^T\,\boldsymbol{\sigma}^e\,d\,\Omega \tag{2}$$

and then assembled by

$$\boldsymbol{f}_{int} = \sum_{e} \boldsymbol{L}^{e^T}\,\boldsymbol{f}^{e}_{int} \tag{3}$$

where,

$\boldsymbol{f}^e_{int}$	$=$	element internal nodal force,
Ω^e	$=$	domain of the element,
$\boldsymbol{B}$	$=$	gradient matrix,
$\boldsymbol{\sigma}^e$	$=$	Cauchy stress matrix,
$\boldsymbol{L}^e$	$=$	Boolean connectivity matrix.

Equation 3 gives the assembly of the element internal nodal forces into the global array. The array $\boldsymbol{L}^e$ is never computed; instead the operation indicated by Eqn. 3 is implemented by simply adding the entries of the element array into the appropriate locations in the global array as described in Section 3.2.

The element stresses are computed from the corotational components of the velocity strain $\boldsymbol{d}$ given by

$$\hat{d}_{ij} = \frac{1}{2}\left(\frac{\delta\hat{v}_i}{\delta\hat{x}_j} + \frac{\delta\hat{v}_j}{\delta\hat{x}_i}\right) \tag{4}$$

where the superposed 'hat' signifies components expressed in terms of the base vectors of the corotational coordinate system. The velocity at any point in the plate is given by Mindlin theory as

$$\boldsymbol{v} = \boldsymbol{v}^m - \hat{z}\boldsymbol{e}_3 \times \boldsymbol{\theta} \tag{5}$$

where,

$$
\begin{aligned}
\boldsymbol{v}^m &= \text{velocity of plate midsurface,}\\
\hat{z} &= \text{distance from midsurface,}\\
\boldsymbol{e}_3 &= \text{base vector perpendicular}\\
&\quad\ \text{to plane of plate,}\\
\boldsymbol{\theta} &= \text{angular velocity.}
\end{aligned}
$$

Once the corotational components of the velocity strain have be computed, the appropriate constitutive law can be applied to calculate the element stresses.

2.2 Time Integration

The following central difference equations are used to update the nodal velocities and displacements in time. Note that an average time step is used to update the velocities. This allows for the capability of changing the time step during the problem solution.

$$
\begin{aligned}
\boldsymbol{v}^{n+\frac{1}{2}} &= \boldsymbol{v}^{n-\frac{1}{2}} + \Delta t^n \,\boldsymbol{a}^n\\
\boldsymbol{u}^{n+1} &= \boldsymbol{v}^n + \Delta t^{n+\frac{1}{2}} \,\boldsymbol{v}^{n+\frac{1}{2}}\\
\Delta t^n &= \tfrac{1}{2}\left(\Delta t^{n-\frac{1}{2}} + \Delta t^{n+\frac{1}{2}}\right)
\end{aligned}
\tag{6}
$$

where,

$$\begin{aligned} \boldsymbol{u}, \boldsymbol{v}, \boldsymbol{a} &= \text{nodal acceleration, velocity and} \\ &\quad\ \text{displacement, respectively,} \\ \Delta\, t^n &= \text{time increment for step } n. \end{aligned}$$

The superscripts in the above equations designate time steps. The fractional superscripts indicate a midstep value. An outline of the explicit time integration scheme is given below.

Flow Chart for Explicit Integration

1. Initial conditions : $\boldsymbol{v}^{-\frac{1}{2}}, \boldsymbol{x}^{o}$

2. Compute external force

3. Compute internal force vector $\boldsymbol{f}_{int}^{n+1}$

   Loop over element blocks

   (a) compute velocity strains

$$\boldsymbol{d}^{n+\frac{1}{2}} = \boldsymbol{B}\, \boldsymbol{v}^{n+\frac{1}{2}} \qquad (7)$$

   (b) compute frame invariant stress rates

$$\overset{\nabla}{\boldsymbol{\sigma}}{}^{n+\frac{1}{2}} = \mathcal{S}(\boldsymbol{\sigma}, \boldsymbol{d}) \qquad (8)$$

(c) convert frame invariant rate to time derivative of Cauchy stress

$$\overset{\nabla}{\boldsymbol{\sigma}}^{n+\frac{1}{2}} = \dot{\boldsymbol{\sigma}}^{n+\frac{1}{2}} - \boldsymbol{W} \cdot \boldsymbol{\sigma}^{n+\frac{1}{2}} + \boldsymbol{\sigma}^{n+\frac{1}{2}} \cdot \boldsymbol{W} \tag{9}$$

(d) update stress

$$\boldsymbol{\sigma}^{n+1} = \boldsymbol{\sigma}^{n} + \Delta t\, \dot{\boldsymbol{\sigma}}^{n+\frac{1}{2}} \tag{10}$$

(e) compute element internal nodal force : Eqn. 2

(f) assemble into global array

4. Compute accelerations by equation of motion : Eqn. 1

5. Update velocities and displacements using central difference equations: Eqn. 6

6. Go to 2.

REMARK: In Eqn. 8, $\overset{\nabla}{\sigma}$ is a frame invariant rate such as the Jaumann rate and $\boldsymbol{W}$ is the spin tensor.

2.3 Evaluation of Critical Time Step

For explicit problems, the time step is calculated based on a numerical stability criterion. The critical time step for a undamped linear system of equations updated using central difference equations is given by [3]:

$$\Delta t_{cr} = \frac{2}{\omega_{max}} \tag{11}$$

where ω_{max} is the maximum frequency of the system

$$\mathbf{Ku} = \lambda \mathbf{Mu} \tag{12}$$

where,

$$\lambda = \omega_{max}^2 \tag{13}$$

The element eigenvalue inequality theorem states that the maximum absolute system eigenvalue is bounded by the maximum element eigenvalue, i.e.,

$$|\lambda| \leq |\lambda_{E_{max}}| \tag{14}$$

where,

$$\lambda \quad = \quad \text{maximum system eigenvalue,}$$

and,

$$\lambda_{E_{max}} \quad = \quad \text{maximum } \lambda_e \text{ for all elements.}$$

The maximum frequency for a one dimensional rod element with linear displacements and diagonal mass can be easily calculated as $2c/l$ where c is the dilatational wave speed and l is the length of the element. The critical time step for the element is l/c . Physically, this time step corresponds to the amount of time required for a stress wave to traverse stress dimension of the element. Therefore, the critical time step for explicit time integration is calculated based on the dimensions and material properties of the element with the largest frequency. The critical time step decreases as the size of the element decreases.

Frequencies for the bending, shear and membrane response of the 4-node Mindlin plate element are presented in [4, 5] and summarized below. The critical time step of the element corresponds to the largest of the computed frequencies.

Bending:

$$\omega_{max} = \left[\frac{\frac{D}{A} \left\{ R_1 + [R_1^2 - 4\,(1-\nu^2)\;R_2^2]^{\frac{1}{2}} \right\}}{\rho\, A\, h\, \alpha} \right]^{\frac{1}{2}} \tag{15}$$

Membrane:

$$\omega_{max} = \frac{1}{A} \left\{ \frac{E}{(1-\nu^2)\,\rho} \left[R_1 + \left(R_1^2 - 4\,\left(1-\nu^2\right)\; R_2^2 \right)^{\frac{1}{2}} \right] \right\}^{\frac{1}{2}} \tag{16}$$

Shear:

$$\omega_{max} = \left[\frac{\frac{4\,c\,\alpha}{A}\,\hat{a}_{11} + c\,A}{\rho\, A\, h\, \alpha} \right]^{\frac{1}{2}} \tag{17}$$

where,

$$
\begin{aligned}
D &= \tfrac{E h^3}{12(1-\nu^2)}, \\
\alpha &= \tfrac{h^2}{12}, \\
\hat{a}_{11} &= \tfrac{1}{4}(R_1 + R_3), \\
R_1 &= y_{24}^2 + y_{31}^2 + y_{24}^2 + y_{31}^2, \\
R_2 &= y_{32}\, x_{24} - y_{24}\, x_{31}
\end{aligned}
$$

and,

$$R_3 = (R_1^2 - 4\, R_2^2)^{\frac{1}{2}}.$$

The stability analysis performed to estimate the critical time step is based on a linear system of equations. However, experience has shown that a linearized analysis provides good estimates of the stable time step. For nonlinear problems, the critical time step is reduced 5% to 10% to compensate for potential destabilizing effects due to nonlinearities. In addition, an energy balance is performed for every time step in order to monitor the stability of the system.

3 Vectorization

3.1 Compiler Vectorization

When compiling a program on a computer with vector processors, options are available for automatic vectorization. The compiler will attempt to vectorize each do-loop in the program. Compilers differ in their ability to vectorize programming constructs such as IF statements in loops. However, current compilers will not vectorize do-loops which contain any of the following statements:

1. Data dependencies
2. Ambiguous subcripts
3. Certain IF statements such as
   (a) block IF, ELSE, ENIDIF with nesting at a level greater than 3
   (b) ELSE IF statements
   (c) IF, GOTO label outside of loop
4. READ or WRITE statements
5. Subroutine calls

The compiler will usually issue an explanation if it is unable to vectorize the do-loop. Additional details about vectorization can be found in [6].

In order to maximize the benefits of vectorization, modifications to the program are frequently required. In many cases, minor changes are sufficient to enable a do-loop to vectorize or to improve the efficiency of the do-loop. The following examples illustrate two typical situations in which an existing do-loop can be easily modified for efficient vectorization.

In nested do-loops, only the innermost do-loop will vectorize. Therefore, the inner do-loop should have the largest range of indicies. If this is not the case, the inner and outer loops can sometimes be interchanged without affecting the calculations. If the range of the inner do-loop is sufficiently small, the inner loop can be "unrolled," thus allowing the outer do-loop to vectorize. In the following do-loop, the compiler will attempt to vectorize the inner loop, leaving the remaining calculations to be performed in scalar mode.

```
      DO 10 I = 1,1000
                .
                .     (other calculations)
                .

        A(I) = I*I
        DO 10 K = 1,3
          A(I) = A(I) + B(K)
   10 CONTINUE
```

Unrolling the inner do-loop allows all calculations to vectorize.

```
      DO 10 I = 1,1000
                .
                .     (other calculations)
                .

        A(I) = I*I
        A(I) = B(1) + B(2) + B(3)
   10 CONTINUE
```

Programming techniques are frequently different for vectorized codes than for scalar codes. For scalar programs, efficient coding consists of minimizing the number of calculations performed. In a vectorized code, it is more important to retain the vector structure of the computations. For example, in the following scalar loop, it is worthwhile to use an IF statement to check whether the component of A is equal to zero and if it is, omit the computation. A GO TO statement avoids unnecessary calculations.

```
      DO 10 I = 1,1000
        IF (A(I) .EQ. 0.0) GO TO 10
        D(I) = D(I) + A(I)*C(I)/(I*I)
   10 CONTINUE
```

In a vectorized version of this loop, it is important to eliminate the IF statement to preserve vectorization.

Suppose that the IF statement in the above example read

```
IF (A(I) .GE. 3.2*B(I)) GO TO 10
```

It is no longer possible to simply remove the IF statement. Some compilers will vectorize this type of do-loop by doing a *gather/scatter* operation on the vector A. In gather/scatter, the compiler creates a temporary array which contains all values of A(I) less than 3.2 times B(I) and computes the update on D(I) only for this subset of A. If the compiler does not have a gather/scatter capability, it is possible to maintain vectorization by defining a temporary vector for the calculation.

```
      DO 10 I = 1,1000
        TEMP(I) = A(I)
   10 CONTINUE

      DO 20 I =1,1000
        IF (A(I) .GE. 3.2*B(I)) TEMP(I) = 0.0
   20 CONTINUE

      DO 30 I = 1,1000
        D(I) = D(I) + TEMP(I)*C(I)
   30 CONTINUE
```

In the preceding loops, only the first and third loops will vectorize for compilers without gather/scatter capabilities. The first loop is overhead required to retain vectorization in the third loop. The second loop is performed in scalar mode. Although this example is rather trivial, the technique can be quite useful for vectorizing many types of loops. As will be discussed in the section on concurrency, creating temporary arrays also helps minimize memory contention problems inherent in shared-memory multiprocessors.

Minor modifications, such as those presented above, will yield moderate improvements in speed-up due to vectorization. However, in order to best exploit the vectorization capabilities of the computer, it is frequently necessary to restructure the flow of the program by replacing calculations for a single element or node by loops which perform the calculations for a group of elements or nodes. This restructuring is discussed in the following section.

3.2 Vectorization of Internal Nodal Force Array

One way to approach the vectorization of a large program is to determine which portions of the code require the most computational time. The longer the computational time, the more effort should be devoted to vectorization. For an explicit finite element program, a large majority of the time is consumed by the computation of the internal nodal force vector, $\boldsymbol{f}_{int}$. In the scalar code, the element internal force vector is calculated and assembled into the global array for one element at a time. Since the internal force vectors of all elements are independent at a given time step, the internal force calculations are very conducive to vectorization. Instead of performing the calculations for individual elements, the internal force computation can easily be vectorized by placing the operations within a loop and performing the calculations for a block of elements.

The procedure is as follows: The elements are divided into blocks of identical element type and material model. It does not matter if material properties for each element are identical as long as the model (i.e., elastic or von Mises elastic-plastic) is the same. The number of elements placed in each block depends on the length of the vector registers and certain characteristics such as problem size. The criteria for selecting block size are discussed later.

Once the elements have been divided into blocks, the scalar calculations can be transformed to vector calculations by converting scalar variables to arrays and placing operations in do-loops. For example, trial stresses for an elastic-plastic material model are computed in scalar mode for one element by:

```
SNEW1 = SOLD1 + SDEL1
SNEW2 = SOLD2 + SDEL2
SNEW3 = SOLD3 + SDEL3
```

In vectorized form, the calculations are modified as:

```
   DO 10 J = 1,NEPB
     SNEW1(J) = SOLD1(J) + SDEL1(J)
     SNEW2(J) = SOLD2(J) + SDEL2(J)
     SNEW3(J) = SOLD3(J) + SDEL3(J)
10 CONTINUE
```

where NEPB is the number of elements in the block. The computed arrays are stored in common blocks so that they can be accessed by any subroutine. Note that vectorization substantially increases the amount of memory required because of the large number of arrays that are created. In older computers, the small core capacity would have significantly limited the size of problems which could run using a vectorized code. However, recent technological advances have made large memory cores available and practical, thus eliminating size limitations except for extremely large problems.

Vectorization is fairly straightforward for many computations, however certain modifications must be made to exploit vectorization in some algorithms. The calculation of the plastic constitutive equation is an illustration of this situation. In a scalar code, a trial stress state is computed for the element and compared to the yield stress. If the element is elastic, the stresses are updated and the subroutine is exited. However if the element is plastic, additional calculations are required. In a vectorized code, the same calculations must be performed for all elements in the block. When a block contains a mixture of elastic and plastic elements, the elastic elements must perform the plastic calculations without modifying the elastic stresses. This was accomplished by creating two arrays, KE(NEPB) and KP(NEPB) which indicate whether the element is elastic or plastic (KE = 1 and KP = 0 if the element is elastic and visa versa if the element is plastic). If all elements are elastic, the stresses are updated and the plastic calculations are omitted. Otherwise, all elements perform the plastic stress calculations and the appropriate stress is stored. The following coding illustrates the flow of the vectorized calculations of the updated stresses.

```
C
C     COMPUTE TRIAL STRESS STATE
```

```
C
      DO 10 J = 1,NEPB
        SNEW1(J) = SOLD1(J) + SDEL1(J)
        SNEW2(J) = SOLD2(J) + SDEL2(J)
        SNEW3(J) = SOLD3(J) + SDEL3(J)
C
C     APPLY YIELD CRITERION
C     IF ELASTIC : KP = 0, KE = 1
C     IF PLASTIC : KP = 1, KE = 0
C
        S1(J) = SQRT(SIGEF2(SNEW1(J),SNEW2(J),SNEW3(J)))
        KP(J) = 0.5 + SIGN(0.5,S1(J) - YIELD(J))
        KE(J) = 1. - KP(J)
 10   CONTINUE
C
C     IF ALL ELEMENTS ARE ELASTIC, RETURN
C
      DO 20 J = 1,NEPB
        IF (KE(J) .EQ. 0) GO TO 40
 20   CONTINUE
      RETURN
 40   CONTINUE
C
C     COMPUTE PLASTIC STRESS
C
      SPL1(J) = . . .
      SPL2(J) = . . .
      SPL3(J) = . . .
C
C     UPDATE PLASTIC STRESS FOR PLASTIC ELEMENTS AND
C     ELASTIC STRESS FOR ELASTIC ELEMENTS
C
      DO 60 J = 1,NEPB
        SNEW1(J) = KP(J)*SPL1(J) + KE(J)*SNEW1(J)
        SNEW2(J) = KP(J)*SPL2(J) + KE(J)*SNEW2(J)
        SNEW3(J) = KP(J)*SPL3(J) + KE(J)*SNEW3(J)
 60   CONTINUE
```

REMARK 1: The DO 20 loop will not vectorize because it contains a GO TO statement to a label outside of the loop.

REMARK 2: Radial return is a particularly simple plasticity algorithm that is easily vectorized. However, radial return is not readily adapted to plane stress.

Some calculations, such as those containing data dependencies, should not be vectorized. Most compilers will check for data dependencies and automatically supress vectorization. However, a compiler directive which prevents vectorization can also be placed immediately preceding the location where vectorization should be stopped. Options are available to enforce the directive for a single loop, for the rest of the subroutine or for the rest of the program. Data dependencies occur frequently when nodal arrays stored in global memory are updated. For example, after the internal nodal forces for an element is computed, it must be assembled into the global array. In vectorized form the assembly procedure for the x-component of the element internal force is:

```
   DO 10 J = 1,NEPB
     FINT(N1(J) + 1) = FINT(N1(J) + 1) + F1X(J)
     FINT(N2(J) + 1) = FINT(N2(J) + 1) + F2X(J)
     FINT(N3(J) + 1) = FINT(N3(J) + 1) + F3X(J)
     FINT(N4(J) + 1) = FINT(N4(J) + 1) + F4X(J)
10 CONTINUE
```

where,

N1,N2,N3,N4	=	Shared memory location indices for local nodes 1,2,3 and 4,
FINT	=	Global internal nodal force vector,
F1X,F2X,F3X,F4X	=	Internal nodal force increment for local nodes 1,2,3 and 4 of element J.

If this loop were allowed to vectorized, the pipeline processor will first retrieve from memory the values of the internal nodal force vector for local node 1 of all elements in the block. These values are stored in a vector register. The temporary array F1X, which contains the updates for each node, is added to the internal nodal force vector. The result is then replaced in global memory. An error would arise when two elements in a block have the same global node for local node 1. For example, suppose elements 1 and 3 have global node 35 as their local node 1. The internal force increment for both element 1 and 3 will be added to the same value of the internal force of node 35. However, when the updated value for node 35 is returned to memory, only the contribution from element 3 is saved. The update from element 1 is stored first and then overwritten by the update from element 3. Vectorization must be prevented in all loops containing updates to nodal variables in the element internal force calculations. It is not necessary to inhibit vectorization in arrays which pertain to element variables such as stress, strain and thickness because there will be no data dependencies among elements in a block.

Techniques have been developed to avoid data dependencies when updating arrays stored in global memory such as the internal force vector discussed above. In [7], an algorithm is presented which divides elements into blocks based on the criterion that no two elements in a block share a common node. This algorithm eliminates the data dependencies in the update of the nodal array and allows the loop to vectorize. Note, that a gather-scatter operation is still required to update the array because of the nonconstant stride between entries stored in global memory. Therefore, for a simple update of a globally stored array, an algorithm eliminating data dependencies will not yield significant speed-up. However, if the elements of the nodal array are used for additional computations, such as the matrix multiplication presented in [7], substantial benefit can be achieved.

4 Concurrency

Concurrency can be implemented using compiler options for all calculations except for the assembly of the internal force vector. In compiler implementations of vectorization and concurrency, loops whose indices exceed the length of the vector registers will be executed in vector-concurrent mode. However, an effective implementation of vectorization-concurrency requires reprogramming with *monitors* which allow the scheduling of calculations among processors.

Two monitors were used for the parallelization of the code. The *askfor* monitor controls the assignment of tasks to the available processes. There are two types of processes. The master process is created by the operating system and performs all of the scalar operations as well as part of the parallel operations. The slave processes are created by the master process for parallel computations only. The task assigned to each process is to compute the internal force vector for one block of elements. Note that processes involve blocks of elements because of vectorization. Prior to entering the parallel operations, two macros are called by the master process. The first, *probstart*, initializes the task number. The task number refers to the block of elements to be assigned to a process.

The second macro, *create_and_work*(NPROCS) creates NPROCS-1 slave processes, where NPROCS is the number of competing processes. NPROCS also corresponds to the number of available processors and is an input variable. Each of the slave processes calls SUBROUTINE WORK, which is the subroutine which invokes the askfor monitor. The master process then calls SUBROUTINE WORK. Therefore, NPROCS processes are executing the operations in WORK simultaneously. The create_and_work macros is defined by:

```
          define(create_and_work,
            [DO 30 I = 1,NPROCS-1
               create(SLAVE)
      30     CONTINUE
             CALL WORK ] )
```

In SUBROUTINE WORK, the askfor monitor is invoked using the following expression.

```
askfor (MO,RC,NPROCS,getsub(I,NBLOCKS,RC),reset)
```

The master process enters the askfor monitor with SUB = 1, the current task number. SUB is the shared data and is initialized to 1 by the macro *probstart.* The macro *getprob* assigns the task number to the process in the monitor. The subcript SUB is then incremented and the return code RC is set to 0 indicating a successful acquisition of the task. The process exits the monitor allowing the next process to enter. This procedure continues until the incremented subscript exceeds NBLOCKS, in which case the processes are delayed. When all slave processes are delayed, the master process exits the monitor operations and returns to the nonparallelized code.

The second monitor used is the *lock* monitor which is used to protect access to shared memory. When variables from shared memory are required for parallel calculations, they are first stored in temporary arrays using a "gather" operation discussed in Section 3.1. This minimizes memory contention problems encountered in shared-memory multiprocessors and also benefits vectorization. However, if two or more processes attempt to access the same memory location simultaneously, an error will occur. Therefore, the gather operation is placed within a lock monitor and only one process is allowed to access a particular subset of memory at a time. In other words, the instruction to access global memory becomes the monitor operation.

A lock is invoked by a *lock* macro immediately preceding the operation. The locks are named so that different subsets of memory can be associated with different lock monitors. Control of the monitor is released after the operation by the *unlock* macro. The following example shows the "gathering" of the x-coordinates of the nodes in a quadrilateral plate element into temporary arrays labeled X1, X2, X3 and X4. The coordinates of the nodes are stored in shared array AUX and the nodal locators are stored in local arrays N1, N2, N3 and N4 for nodes 1, 2, 3 and 4, respectively. L1 is the name of the lock which is associated with the memory locations containing the x-coordinates of all nodes. LL2(NID) is the number of elements in the block assigned to process NID.

```
           nlock(L1)
           DO 10 J = 1,LL2(NID)
             X1(J,NID) = AUX(N1(J,NID) + 1)
             X2(J,NID) = AUX(N2(J,NID) + 1)
             X3(J,NID) = AUX(N3(J,NID) + 1)
             X4(J,NID) = AUX(N4(J,NID) + 1)
      10   CONTINUE
           nunlock(L1)
```

Several named locks are used; each corresponding to a different component of the nodal arrays used in the internal force calculations. By applying a lock to each component, the number of operations within a given lock monitor can be reduced, thus minimizing slowdown due to the locks. Note that the components of the nodal arrays are retrieved from global memory only once during the element block calculations. After they have been stored in temporary arrays, memory contention problems are eliminated.

Note also that in this example, the one dimensional arrays created for vectorization have been converted to two dimensional arrays. The added dimension is required to create a local memory for each process. The askfor monitor assigns an identification number NID to the process which indicates which process is performing the operations. The identification numbers range from 1 through the number of competing processes used in the problem solution. The askfor monitor then assigns a block of elements to each available process. Each process will perform the same calculations for different data. By dimensioning the temporary arrays as TEMP(NEPB,NPROCS) where NEPB is the number of elements per block and NPROCS is the number of processes, unique memory is created for each process.

5 Numerical Studies

Numerical studies were made to determine the speed-up possible on a multiprocessor due to both vectorization and concurrency. For comparisons, three version of the program were used:

1. the original version of WHAMS run in scalar, serial mode : WHAM0,
2. the original version of WHAMS compiled using full optimization for concurrency and vectorization : WHAM_OPT
3. the vectorized, parallelized version of WHAMS using both full compiler optimization and monitors to control concurrency in the element calculations : WHAMS_VECPAR.

Three problems were considered:

1. a spherical cap loaded by a uniform pressure;
2. a pressurized containment vessel with a nozzle penetration;
3. an impulsively loaded cylindrical panel.

Results are presented in terms of the total run time for the problem and analyzed by the *speed-up* and *efficiency* of the program. Speed-up is defined as the ratio of the computing time of the program on a serial machine to the computing time on a parallel machine. The efficiency of the program is defined as the speed-up divided by the number of processors. Speed-up and efficiency are strongly influenced by the degree of parallelism and vectorization achieved in the program.

The first problem is the spherical cap shown in Figure 1. The material properties and parameters are listed in Table 1. The problem has 91 nodes and 75 elements and was run primarily to ensure that the vectorized version of the code gave the same results as the nonvectorized code. Because the problem is small, the elements were divided into 8 blocks in order to maximize the benefits of parallelization. However, with only 10 elements per block, the benefits due to vectorization were diminished. Table 2 compares the execution times of the three versions of WHAMS run on the Alliant FX/8 as well as execution times of WHAM0 on the VAX 11/780 and IBM 3033. A speed-up of 11 was achieved by the VECPAR version of WHAMS when compared with the scalar version WHAM0 on the Alliant.

The second problem studied was a pressurized containment vessel with a nozzle penetration shown in Figure 2. The problem has 344 elements and 407 nodes, and is subjected to a uniform pressure. The material properties and mesh dimensions are presented in Table 3 and timings are shown in Table 4.

Comparison of the run times of WHAM0 and WHAM_OPT shows that using compiler optimization for concurrency and vectorization provides a speed-up of almost three. This is only 3/8 of the speed-up which should be achieved by concurrency alone. However, by vectorizing the code and using monitors to control parallelism of the internal force vector, a total speed-up of 18.7 was achieved.

Three element block sizes were used for the WHAM_VECPAR version of the code using 1, 4 and 8 processors. Efficiencies due to parallelization were calculated by comparing run times using multiple processors with those using a single processor. Using four processors, the efficiencies for 12, 24 and 32 elements per block were 81%, 78% and 77%, respectively. With eight processors, the efficiencies decreased to 66%, 60% and 51%, respectively. The trend indicates that efficiencies decrease as the number of processors increase and as the number of elements per block increase. Note, however, that the efficiency due to vectorization increases as the number of elements per block increases. Therefore, a trade off exists between optimizing a code for vectorization and concurrency. These trends will be discussed in further detail in the next section.

The final problem is a 120 degree cylindrical panel shown in Figure 3 which is hinged at both ends and fixed along the sides. The panel is loaded impulsively with an initial velocity of 5650 in/sec over a portion of the shell. An elastic-perfectly plastic constitutive model was used with four integration points through the thickness. The material properties are shown in Table 5. Further details can be found in [8]. Due to symmetry only half of the cylinder was modeled. Three different uniform mesh discretizations were used so that the effects of problem size and element block size could be studied. Table 6 shows the number of elements and nodes for each mesh as well as time step used and total number of time steps.

All mesh discretizations were run using the three versions of WHAMS described previously. The results are presented in Tables 7 through 9. The VECPAR version of the program was run using 1,4 and 8 processors and various element block sizes.

A comparison of run times between the original version of the code, WHAM0, and the code using compiler optimization, WHAM_OPT, shows a speed-up of approximately 2.5 for all mesh discretizations. Comparing the original version of the code with the VECPAR version using eight processors gives speed-ups of 17.4, 24.2 and 26.4 for mesh 1, 2 and 3, respectively. Total speed-ups increase as the problem size increases.

Full advantage of vectorization on the Alliant FX/8 can be taken by using vectors of at least 24 elements in length. The optimum vector length is 32 which is the size of the vector registers. As the number of elements per block increases, the run times decrease due to vectorization. However, the efficiency attributed to concurrency also decreases. In going from 1 processor to 4, the average speed-up is 3.53 with an efficiency of 88%, while with 8 processors, the average speed-up is 5.70 with an efficiency of 71%.

Assigning one element to a block eliminates the benefits of vectorization in the internal force calculations. The average speed-up achieved for this case using 4 and 8 processors was 3.92 and 7.55, yielding efficiencies of 98% and 94%, respectively. Run times for one element per block were quite high illustrating the fact that the execution of vectorized do-loops with only a few loop iterations is slower than performing the operations in scalar mode.

6 Summary of Results

The decrease in efficiency is caused by factors inherent in the design of parallel algorithms. One factor is that processors remain idle when the number of tasks is smaller than the number of available processors. For example, the internal force calculations for 8 blocks of elements will take the same amount of time as 1 block of elements if eight processors are available. In the latter case, seven of the processors will be idle while the eighth performs the computations. The problem of processor idleness also illustrates an advantage to using relatively small element blocks (24 to 32 elements/block) as opposed to a few very large blocks. The more blocks that are available for computation, the less likely a processor will remain idle. Also, the larger the problem, the less significant processor idleness becomes. In terms of storage requirements, smaller block size is also preferable.

Another factor which decreases efficiency is memory contention. If more than one processor attempts to access a shared memory location simultaneously, an error will occur. This happens in internal force calculations when two elements in different blocks share a node. A "lock" monitor must be used to ensure that only one processor will access the memory location at a time. However, the locks may create a slowdown if substantial interference exists.

Probably the most significant factor for the decreasing efficiency with the number of processors is the effect of nonparallelizable computations. Once the most time consuming computations have been effectively recoded for parallel computations, other portions of the program require an increasingly larger fraction of the computation time. These sections of the code may not be conducive to parallel execution and will prevent further speed-up.

To illustrate this effect, the program was divided into three parts: the calculations performed before, during and after the element internal force computation. The first part included the calculation of the external force array as well as the update of the nodal coordinates. The second part was comprised of all calculations listed in step 3 of the flow chart in Section 2.2. The final section included the computation of the accelerations and the update of the velocity and displacement vectors. The times required for each section was monitored for the cylindrical panel problem with 24 elements per block and are presented in Table 10 for 1, 4 and 8 processors. The speed-up (efficiency) calculated for the internal force calculations was 6.1 (76%), whereas the speed-up (efficiency) of the computations before and after these calculations was 2.5 (31%) and 4.7 (60%), respectively. The average speed-up and efficiency for the time step was 5.8 and 73%, respectively. Furthermore, as the number of processors increase, the percentage of time spent in the internal force calculations decreases slightly. Therefore, the less efficient coding takes up an increasingly greater percentage of the total execution time.

Table 1: Material Properties and Parameters for Spherical Cap

Radius	r	=	22.27 in
Thickness	t	=	0.41 in
Angle	α	=	26.67 deg
Density	ρ	=	2.45×10^{-4}lb-sec^2/in^4
Young's modulus	E	=	1.05×10^7 psi
Poisson's ratio	ν	=	0.3
Yield stress	σ_y	=	2.4×10^4 psi
Plastic modulus	E_p	=	2.1×10^5 psi
Pressure load	P	=	600 psi
Time Steps	N	=	1000

Table 2: Solution Times for Spherical Cap Problem

Alliant - WHAM0	310.9	sec
Alliant - WHAM_OPT (8 Procs.)	116.5	sec
Alliant - WHAM_VECPAR (8 Procs.)	28	sec
VAX 11/780 - WHAM0	901.8	sec
IBM 3033 - WHAM0	75	sec

Table 3: Material Properties and Parameters for Containment Vessel

Vessel diameter	d_v	=	264.0 in
Vessel height	h	=	399.0 in
Penetration diameter	d_p	=	40.0 in
Penetration length	l	=	29.3 in
Thickness	t	=	0.25 in
Density	ρ	=	7.5×10^{-4}lb-sec^2/in^4
Young's modulus:			
Nozzle	E	=	40.0×10^7 psi
Pressure vessel	E	=	3.0×10^7 psi
Collar	E	=	9.0×10^7 psi
Poisson's ratio	ν	=	0.3
Yield stress	σ_y	=	6.01×10^4 psi
Plastic modulus	E_p	=	4.4×10^4 psi

Table 4: Run Times (Efficiency) for Containment Vessel Problem

Program Version	Number of Processors	12 elements per block	24 elements per block	32 elements per block
WHAM0	1	3768		
WHAM_OPT	8	1291		
WHAM_VECPAR	1	1167	959	907
	4	356(81%)	309(78%)	295(77%)
	8	222(66%)	201(60%)	223(51%)

Table 5: Parameters for Cylindrical Panel Problem

Density	ρ	=	2.5×10^{-4}lb-sec^2/in^4
Young's modulus	E	=	1.05×10^7 psi
Poisson's ratio	ν	=	0.33
Yield stress	σ_y	=	4.4×10^4 psi
Plastic modulus	E_p	=	0.0 psi

Table 6: Sizes and Time Steps for Mesh Discretizations for Cyl. Panel Problem

Mesh	No. Elements	No. Nodes	Time Step	No. Steps
1	96	119	2.0E-6 sec	500
2	384	429	1.0E-6 sec	1000
3	1536	1625	0.5E-6 sec	2000

Table 7: Run Times (Efficiency) for Cylindrical Panel Problem - Mesh 1

Program Version	Number of Processors	1 element per block	12 elements per block
WHAM0	1	347	
WHAM_OPT	8	141	
WHAM_VECPAR	1	529	103
	4	137(97%)	31(83%)
	8	74(89%)	20(64%)

Table 8: Run Times (Efficiency) for Cylindrical Panel Problem - Mesh 2

Program Version	Number of Processors	1 element per block	12 elements per block	24 elements per block	32 elements per block
WHAMO	1	2658			
WHAM_OPT	8	1072			
WHAMO_VECPAR	1	4189	785	631	594
	4	1071(98%)	213(92%)	176(90%)	168(88%)
	8	552(95%)	126(78%)	110(72%)	125(59%)

Table 9: Run Times (Efficiency) for Cylindrical Panel Problem - Mesh 3

Program Version	Number of Processors	1 element per block	12 elements per block	24 elements per block	32 elements per block
WHAMO	1	20860			
WHAM_OPT	8	8484			
WHAMO_VECPAR	1	34495	6030	4807	4496
	4	8470(100%)	1671(90%)	1391(86%)	1275(88%)
	8	4364(99%)	939(80%)	812(74%)	789(71%)

Table 10: Comparison of Computation Times For a Single Time step

Section	1 Proc.	4 Procs.	8 Procs.	Speed-up
Before f_{int}	.0032	.0017	.0013	2.46
During f_{int}	.5364	.1483	.0883	6.07
After f_{int}	.0714	.0224	.0150	4.76
Total	.6110	.1724	.1046	5.84

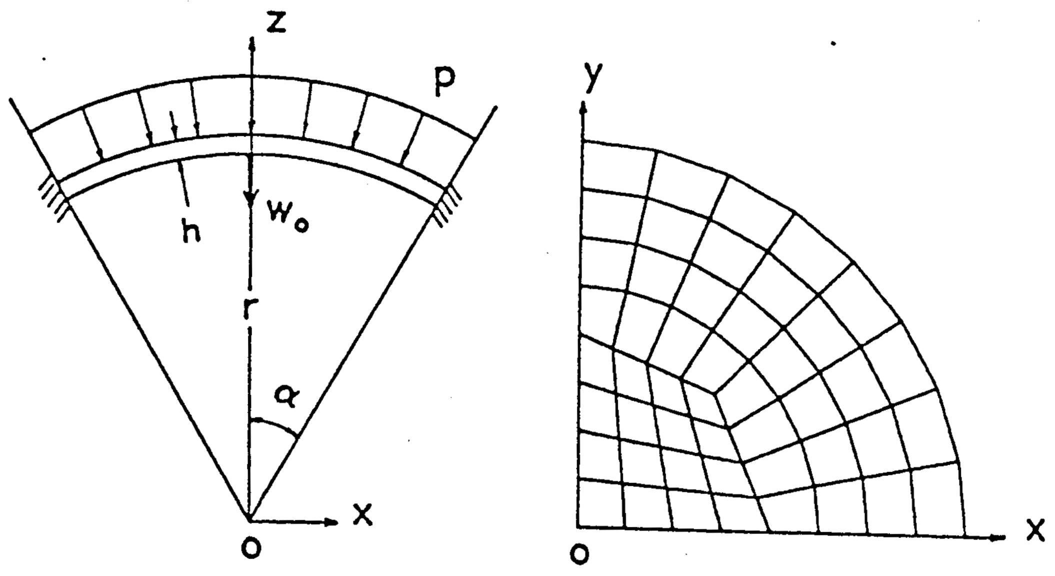

Figure 1: Sample Problem 1: Spherical Cap

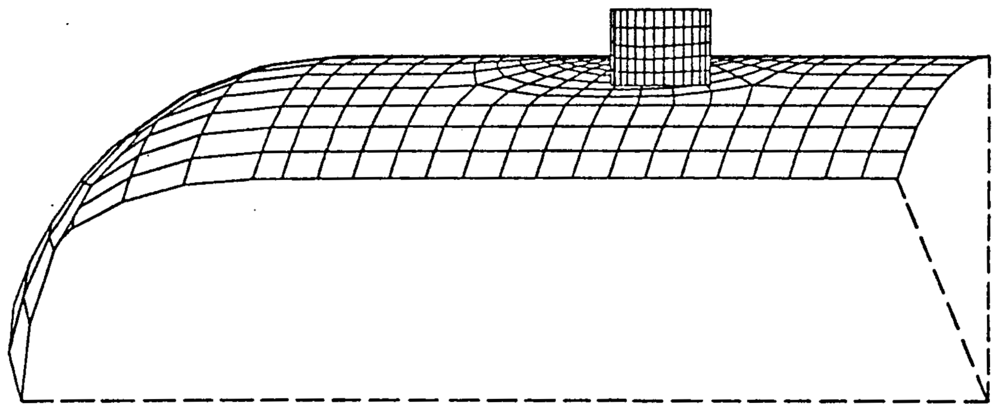

Figure 2: Sample Problem 2: Containment Vessel with Nozzle Penetration

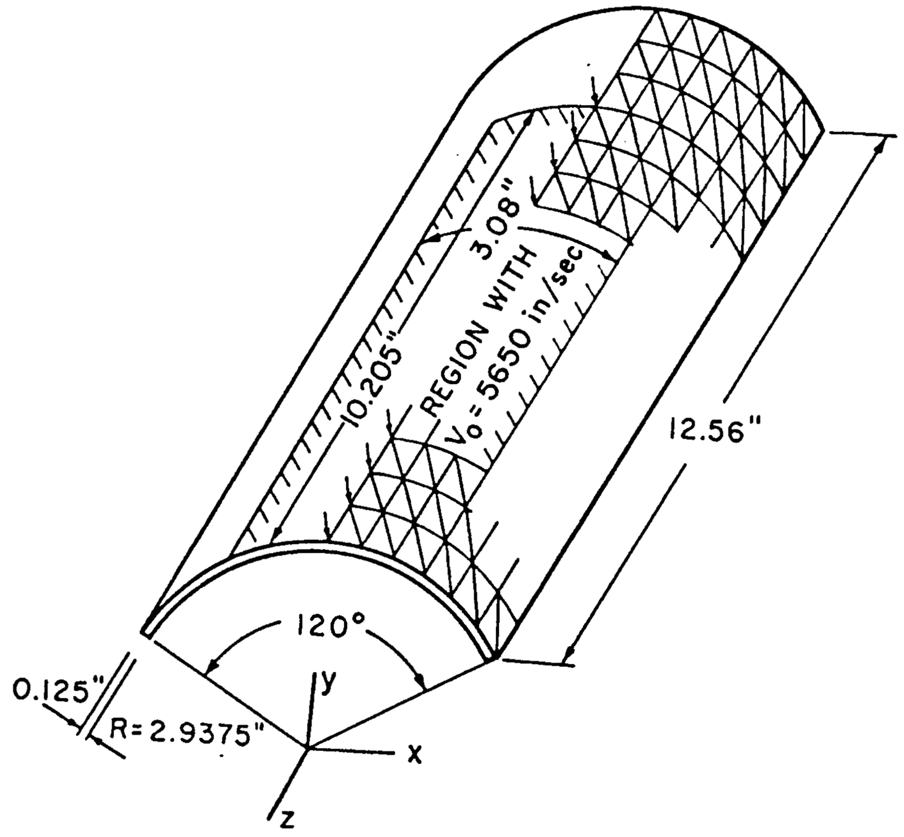

Figure 3: Sample Problem 3: Impulsively Loaded Panel

References

[1] Belytschkko, T., Tsay, C. S., "WHAMSE: A Program for Three-Dimensional Nonlinear Structural Dynamics," EPRI Report NP-2250, Palo Alto, CA, February 1982.

[2] Yamada, Y., Yoshimura, N., and Sakurai, T., "Plastic Stress-Strain Matrix and its Applications for the Solution of Elastic-Plastic Problems by the Finite Element Method," *International Journal of Mechanical Sciences,* Vol. 10, 1968, pp. 561-578.

[3] Belytschko, T., "Explicit Time Integration of Structural-Mechanical Systems," *Advanced Structural Dynamics,* Donea, J., ed., Applied Science Publishers, Essex, England, 1980, pp. 97-122.

[4] Flanagan, D. P., and Belytschko, T., "Eigenvalues and Stable Time Steps for the Uniform Strain Hexahedron and Quadrilateral," *Journal of Applied Mechanics,* Vol. 51, March 1984, pp. 35-40.

[5] Belytschko, Ted, and Lin, Jerry, I., "Eigenvalues and Stable Time Steps for the Bilinear Mindlin Plate Element," *International Journal for Numerical Methods in Engineering,* Vol. 21, 1985, pp. 1729-1745.

[6] NCSA Training Information

[7] Hughes, Thomas J. R., Ferencz, Robert M., and Hallquist, John O., "Large-Scale Vectorized Implicit Calculations in Solid Mechanics on a CRAY X-MP/48 Utilizing EBE Preconditioned Conjugate Gradients," *Computer Methods in Applied Mechanics and Engineering,* Vol. 61, 1987, pp. 215-248.

[8] Kennedy, J. M., Belytschko, T., and Lin, J. I., "Recent Developments in Explicit Finite Element Techniques and Their Application to Reactor Structures," *Nuclear Engineering and Design,* Vol. 97, 1986, pp. 1-24.